A Dynamic Approach to Termination Analysis for Active Database Rules

James Bailey, Alexandra Poulovassilis, Peter Newson

Department of Computer Science, Birkbeck College, Univ. of London,
Malet Street, London WC1E 7HX, UK. {james,ap,pete}@dcs.bbk.ac.uk

Abstract. An important behavioural property for sets of active database rules is that of termination. In current commercial database systems, termination is guaranteed by imposing a fixed upper limit on the number of recursive rule firings that may occur. This can have undesirable effects such as prematurely halting correct executions. We describe a new approach based on a dynamic upper limit to the number of rule firings. This limit reflects knowledge about past rule behaviour on the database and provides a more accurate measure for when the DBMS should terminate rule execution. The approach incurs little cost and can easily be integrated with current techniques for static analysis of active rules.

1 Introduction

Active databases provide the functionality of traditional databases and additionally are capable of reacting automatically to state changes without user intervention. This is achieved by means of Event-Condition-Action (ECA) rules of the form *on* event *if* condition *do* action. One of the problems of using ECA rules is the inherent difficulty of analysing and controlling their behaviour. An important behavioural property is that of *termination*, since when multiple ECA rules have been defined in an active DBMS, possibly by different people at different times, there is the possibility that the rules may trigger each other indefinitely.

The research community has addressed this problem by focusing principally on static analysis of rule sets. The aim here has been to develop methods which can predict whether a set of rules is always guaranteed to terminate [1, 7, 4]. Since the problem is undecidable even for simple rule languages [3], these methods cannot achieve full precision. Also, given the lack of any consensus on what constitutes a "typical" terminating or non-terminating rule set, it is not easy to compare various methods with one another.

These difficulties are recognised by the current SQL3 standard for triggers [12], which does not attempt to prescribe methods for ensuring termination. Thus, commercial database products do not use any static analysis on rule sets, and termination is enforced by imposing a fixed upper limit (hereafter referred to as k or the k *limit*) on the number of recursive rule firings allowed (in Oracle 8, for example, this limit is 64). If k is reached, the actions of all the rules are rolled back. This approach has the limitation that correct sequences of rule firings may be prematurely halted due to an overly conservative (i.e. too low) choice of k.

Conversely, allowing k to be too high can lead to unnecessary rule processing if a non-terminating computation occurs. This wasted effort may be problematic for applications such as real-time systems, where there are hard deadlines on transaction execution time and where minimising work is important.

In this paper, we describe a new approach to termination management based on a *variable* upper limit on the number of recursive rule firings. Before rule execution begins, some analysis is first performed to calculate a "suitable" k limit for that particular execution. This calculation is performed using information about the past execution behaviour of the rules. The aim is that calculating the k limit dynamically will allow rule executions to run for longer if eventual termination can definitely be predicted. This in turn helps with the second difficulty above, which occurs when the k limit is too high. This is because the rule designer can now choose a lower default k limit, more confident in the knowledge that fewer correct rule execution sequences will be aborted.

The paper is structured as follows. Section 2 defines our assumed rule execution semantics. Section 3 describes our method for inferring future rule termination from past rule execution behaviour using incremental inferencing techniques. Section 4 extends the treatment to detect the possibility of future rule non-termination by checking for repeating states. Section 5 discusses the practical usefulness and applicability of our approach, including how it fits within the overall transaction execution, and how it can be applied to analysis of SQL3 triggers. In Section 6 we compare our work with other approaches and in Section 7 we summarise our results and outline directions for future research.

2 Rule Execution Semantics

We assume that active rules are of the form *on* event if condition *do* action. The event part is of the form $ins(R)$ or $del(R)$, where R is a relation. The condition part is an SQL query. The action part is a sequence of insertions or deletions of the form $\pm R \leftarrow query$ where R is a relation and $query$ an SQL query. In this paper we do not consider UPDATE events as found in SQL3, but the approach described here could easily be extended to cater for these also.

Events occur as transactions execute. A rule is *triggered* when the event specified in its event part occurs. A rule *fires* if it is triggered and its condition part evaluates to true. A rule set is *terminating* if for any initial event and database state, rule processing always terminates. In this paper we will be considering specific execution sequences and are thus interested in whether rule execution terminates from a *given* database state when triggered by a *given* event.

We specify the assumed rule execution semantics as a function, `execRules`. This inputs the current database and an initial schedule of rule actions. It repeatedly pops the first action off the schedule, executes it (by calling `updateDB`), determines which rules have fired as a result (in the `for` loop) and places their actions on the schedule. `execRules` terminates when the schedule is empty [1].

---

[1] We use the notation $[x_1, x_2, \ldots]$ for lists, $< x_1, x_2, \ldots >$ for tuples and $[i]$ for in-

```
function execRules(db:DB,s:Sched):DB;
var  i      : Nat;
     action: Action;
while s != [] do {
     action:= head s;
     s      := tail s;
     db     := updateDB(db,action);
     for i := 1 to NoOfRules do
          if nonEmpty(eval(eventQueries[i],db)) and
             nonEmpty(eval(condQueries[i],db)) then
               s := bind(actions[i],db) ++ s;
}
return(db);
```

Rules are identified by numbers `1..NoOfRules`, in order of increasing priority. A schedule is a list of rule actions waiting execution. We assume the initial update initiating rule execution is itself the action part of one of the rules[2], so the initial schedule passed to `execRules` is a singleton list `[a]` for some action `a`.

A database is a set of pairs of relation names and relation extents. Relations are of three kinds: (i) user-defined ones; (ii) for each user-defined relation R, two *delta* relations, $\triangle R$ and $\bigtriangledown R$; $\triangle R$ ($\bigtriangledown R$) contains tuples inserted into (deleted from) R by the latest action executed; (iii) for each user-defined relation R, two *event* relations, *insEventR* and *delEventR*; *insEventR* (*delEventR*) is non-empty if and only if the latest action executed was an insertion into (deletion from) R.

The types `DB` and `Sched` are defined as follows, where the types `RelName`, `deltaRelName`, `Tuple`, `Query`, `List` and `Set` are self-explanatory:

```
type DB     = Set(RelName,Set(Tuple));
type Action = (DeltaRelName,Query);
type Sched  = List(Action);
```

A number of global variable declarations representing the rules are assumed:

```
var eventQueries:array[1..NoOfRules] of DeltaRelName;
var condQueries :array[1..NoOfRules] of Query;
var actions     :array[1..NoOfRules] of List(Action);
```

`eventQueries` records for each rule the name of the relation which, if it is non-empty, triggers the execution of the rule: if the rule's event part is $ins(R)$ ($del(R)$), this relation will be $\triangle R$ ($\bigtriangledown R$)[3]. `condQueries` represents the condition

dexing array elements. For any list $[x_1, x_2, \ldots]$, *head* $[x_1, x_2, \ldots]$ returns x_1 and *tail* $[x_1, x_2, \ldots]$ returns $[x_2, \ldots]$. ++ is the list concatenation operator.

[2] If not, then the original rule set can be extended with a new rule whose action part is the initial update and whose event part is not triggered by any rule.

[3] This is **semantic** triggering i.e. rules are triggered if delta relations are non-empty. Later we will also consider **syntactic** triggering where rules are triggered when update statements occur, i.e. when event relations are non-empty, regardless of whether the database has changed.

part of each rule, which is just an SQL query. `actions` is the list of updates constituting the action part of each rule. An insertion $+R \leftarrow query$ is represented by the pair $(\triangle R, query)$ and a deletion$-R \leftarrow query$ by $(\bigtriangledown R, query)$.

`execRules` calls four other functions (which are straightforward and so not given here): `updateDB` applies the first action on the schedule to the database and returns the resulting database (including updated delta and event relations). `eval` evaluates a query with respect to the database and returns a set of tuples. `nonEmpty` tests whether a set is non-empty. `bind` takes a list of rule actions and the current database, and substitutes all occurrences of delta relations in the bodies of the rule actions by their current values in the database.

We have assumed *Immediate* rule coupling mode whereby the actions of a fired rule are placed at the front of the schedule[4]. This is the coupling mode assumed by SQL3. However, our method is also applicable to Deferred coupling mode, or mixtures of coupling modes. For reasons of space we do not consider this issue here, and refer the reader to [5] which discusses a general framework for abstract interpretation of active rule execution.

3 Incremental Inference of Termination Behaviour

Our method for inferring future rule termination behaviour is specified below as a function, INC, which performs an incremental inferencing of future rule behaviour using knowledge about past behaviour:

```
function INC(deltaVals:array [1..NoOfRules] of List(TruthVal),
             condVals:array [1..NoOfRules] of TruthVal,
             s:AbstSched):Status;
var  i        : Nat;
     ev, cv   : TruthVal;
     a        : Action;
     eventVals: array [1..NoOfRules] of TruthVal;
while s != [] do {
     <ev,cv,a>:= head s;
     s        := tail s;
     deltaVals:= INFER_DELTAS(deltaVals,deltaQueries,<ev,cv,a>);
     eventVals:= INFER_EVENTS(eventQueries,a,deltaVal_a);
     condVals := INFER_CONDS(condVals,condQueries,a,deltaVal_a);
     for i := 1 to NoOfRules do
        if eventVals[i] != False and condVals[i] != False then
            s := [<eventVals[i],condVals[i],a> | a <- actions[i]]
                 ++ s;
}
return(DefiniteTermination);
```

[4] Since rules are considered in increasing order of priority in the `for` loop, actions of higher priority rules will be placed in front of actions of lower priority ones.

We observe the syntactic similarity between `execRules` and INC. In fact, INC can be considered to be performing an *abstract interpretation* of `execRules` (see [5] for theoretical foundations). The abstract schedule is a list of triples:

```
type AbstSched = List(TruthVal,TruthVal,Action)
```

As well as each action, it records the truth values of the event and condition queries when the rule fired. This is because INC places rules on the schedule if their event and condition queries are inferred to be True or Unknown, and the imprecision caused by the latter case needs to be carried forward into the subsequent inferencing process. Thus, the variables `ev` and `cv` above are the truth values of the rule's event and condition queries when the rule action was placed on the schedule, and `a` is the actual action. The variable `deltaVal_a` denotes the current value of the delta query corresponding to the action `a`.

The database state is abstracted by two arrays which contain the truth values of all the rule conditions and rule delta queries w.r.t. the current database state:

```
var condVals  : array [1..NoOfRules] of TruthVal;
var deltaVals : array [1..NoOfRules] of List(TruthVal);
```

where the type `truthVal` consists of three constants `True`, `False` and `Unknown`.

By *rule delta query* we mean the query that defines the value of the positive or negative delta relation that results when a rule action is executed. In particular, for an insertion $+R \leftarrow query$ the rule delta query is $query - R$ while for a deletion$-R \leftarrow query$ it is $query \cap R$. A rule in general has a list of actions, and hence a list of corresponding delta queries and delta query truth values. We assume the following global variable contains the rules' delta queries:

```
var deltaQueries :array[1..NoOfRules] of List(Query);
```

The `condVals` record whether a rule condition is currently True, False or Unknown. The `deltaVals` record whether a rule action would have a semantic effect (i.e. produce a change in the database) were it to be executed. So, instead of using an actual database with relations containing tuples as does `execRules`, INC uses a coarser view of the database to perform inferencing about how the actual rule execution might proceed.

`condVals` and `deltaVals` need to be maintained in synch with the current database state during actual rule execution. This is done by modifying `execRules` as follows:

(a) When a rule is triggered within the `for` loop and its condition query is then evaluated, the resulting truth value (True or False) will be recorded in the corresponding entry of the `condVals` array.
(b) Condition queries that are not evaluated within the `for` loop (because the rule has not been triggered) will have truth values inferred for them using incremental evaluation techniques (see Section 3.1 for details of these);
(c) New truth values for all the delta queries are similarly inferred on each iteration of the `while` loop.

Notice that under (a) condition queries will be assigned a value of either True or False while under (b) and (c) condition and delta queries may be assigned an Unknown truth value.

INC calls three functions to repeatedly infer the next round of truth values for the rules' delta queries, event queries and condition queries. INFER_DELTAS takes the current values of the delta queries, the delta queries themselves, and the triple `<ev,cv,a>` currently at the head of the schedule and infers new truth values for the delta queries. INFER_EVENTS takes the event queries, the current action `a`, and the current value of the delta query corresponding to `a`, `deltaVal_a`, and infers the truth values of the event queries. Finally, INFER_CONDS takes the current truth values of the condition queries, the condition queries themselves, the current action `a`, and its corresponding delta query value, `deltaVal_a`, and infers the new truth values of the condition queries. The inferencing techniques employed by all three functions are described in Section 3.1 below.

If INC terminates from a given initial action and database state, we can infer that `execRules` would also terminate from that initial action and database state. This is reflected by INC's return status of `DefiniteTermination`.

As it stands, INC may of course fail to terminate. There are several possible ways to force it to do so: place an upper limit on the number of iterations of its `while` loop, or implement a check for the occurrence of repeating states, or both of these. We discuss these methods further in Section 4. With any of these approaches, the status returned by INC would be `PossibleNonTermination`.

3.1 The INFER functions

The three INFER functions invoked by INC use incremental techniques to deduce a new truth value for a query from its old truth value. Suppose we have an event, condition or delta query q and an update occurs in the database. What new truth value will be inferred for q ? If an inclusion occurs into a relation r referenced by q, then the new truth value of q is inferred using the following rules:

Rule Inc1: If the previous truth value of q was T and r doesn't occur in q or occurs only positively, then the new truth value of q is T.
Rule Inc2: If the previous truth value of q was F and r doesn't occur in q or occurs only negatively, then the new truth value of q is F.
Rule Inc3: Otherwise, the new truth value of q is U.

Similarly, if an exclusion occurs from a relation r referenced by q, then the new truth value of q is inferred using the following rules:

Rule Exc1: If the previous truth value of q was T and r doesn't occur in q or occurs only negatively, then the new truth value of q is T.
Rule Exc2: If the previous truth value of q was F and r doesn't occur in q or occurs only positively, then the new truth value of q is F.
Rule Exc3: Otherwise, the new truth value of q is U.

In the above rules, r can be either a user-defined relation or a delta relation since both of these may appear in condition queries and delta queries. For a user-defined relation, inclusions and exclusions will occur via the explicit insertions and deletions specified in the rule actions or in the top-level transaction. For a delta relation, inclusions and exclusions will occur implicitly. In particular, a delta relation $\triangle R$ ($\bigtriangledown R$) will undergo an implicit inclusion whenever an insertion (deletion) occurs on R, and will then undergo an implicit exclusion in the next action executed that is not an insertion (deletion) on R.

An action `a` has a special effect on its own delta query `dq_a` as given by the following rules, where an action $\pm R \leftarrow query$ is *idempotent* if R does not appear within *query*, otherwise it is *non-idempotent*:

Rule Own1: If the old truth value of `dq_a` was F, then the action `a` will not execute and so the new truth value of `dq_a` is also F.

Rule Own2: If `a` is idempotent and the old truth value of `dq_a` was U or T then the new truth value of `dq_a` is F.

Rule Own3: If `a` is non-idempotent and the old truth value of `dq_a` was U or T then the new truth value of `dq_a` is U.

The `INFER_DELTAS` function calculates the new truth value for the delta query, `dq_a`, associated with the action `a` as follows, where `infer_own` performs inferences according to Rules Own1-3 [5]:

```
newDeltaVal_a = ev & cv & infer_own(dq_a,oldDeltaVal_a,a)
```

For all other delta queries `dq_i`, their new truth value is the same as their old truth value if `newDeltaVal_a` is False or if `dq_i` is independent of the action `a` (i.e. the relation in the head of `a` does not appear in `dq_i`). Otherwise, the new truth value of `dq_i` is determined as follows, where `infer_normal` performs inferences according to Rules Inc1-3 and Exc1-3:

```
newDeltaVal_i = ev & cv & infer_normal(dq_i,oldDeltaVal_i,a)
```

The `INFER_EVENTS` function calculates the new truth value for each event query `eq_i` as follows, where `infer_event` returns T if the action `a` matches the event query `eq_i` and F otherwise:

```
newEventVal_i = newDeltaVal_a & infer_event(eq_i,a)
```

Finally, the `INFER_CONDS` function calculates the new truth value for each condition query `cq_i`. The new truth value of `cq_i` is the same as its old truth value if `deltaVal_a` is False or if `cq_i` is independent of `a`. Otherwise the new truth value of `cq_i` is given by:

```
newCondVal_i = newDeltaVal_a & infer_normal(cq_i,oldCondVal_i,a)
```

[5] Here, `&` is a 3-valued logic conjunction operator:

&	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Example 1. Consider the 6 rules below[6]:

Rule	Event	Condition	Action(s)	Delta Query(s)
r_1	$ins(R_4)$	$R_4 \bowtie R_5$	$-R_3 \leftarrow R_8; +R_6 \leftarrow R_4 \bowtie R_5$	$R_3 \cap R_8; (R_4 \bowtie R_5) - R_6$
r_2	$ins(R_6)$	$R_4 - R_3$	$+R_3 \leftarrow R_4 - R_3$	$(R_4 - R_3) - R_3$
r_3	$ins(R_3)$	$R_4 \bowtie R_1$	$-R_1 \leftarrow R_9 \bowtie (R_5 - R_4)$	$(R_9 \bowtie (R_5 - R_4)) \cap R_1$
r_4	$del(R_1)$	$true$	$-R_0 \leftarrow R_5$	$R_0 \cap R_5$
r_5	$del(R_0)$	$true$	$+R_0 \leftarrow R_2 \cap \triangledown R_0$	$(R_2 \cap \triangledown R_0) - R_0$
r_6	$ins(R_0)$	$R_7 \bowtie R_8$	$-R_0 \leftarrow R_3$	$R_0 \cap R_3$

These rules are non-terminating, since r_5 and r_6 could trigger each other infinitely. We will see, however, that knowledge about the initial truth values of the condition and delta queries means that we can predict that the rules will terminate for the current database instance.

Consider the trace given below of successive iterations of the `while` loop in INC, starting from the abstract database state shown in the first line. Recall that a triple <T,T,a> on the abstract schedule indicates an action, a, whose corresponding event and condition queries were both True at the time the action was placed on the schedule. `a_i` denotes the action of rule r_i (for a single-action rule) while `a_i_j` denotes the j^{th} action of rule r_i (for a multi-action rule).

Note that all actions except r_2's are idempotent, so that the delta queries of all other rules are made False by the execution of their actions. Step 1 shows an initial abstract database state where r_1 was triggered. In step 3 r_2 has fired; the deltaVals of r_1 have become False due to the idempotence property of r_1's actions. In step 4 r_3 has fired; the first deltaVal of r_1, the deltaVal of r_5 and the condVal of r_2 have all become Unknown due to the execution of a_2. In step 5 r_4 has fired. In step 6 r_5 has fired. In step 7 r_6 has become triggered; however, since the condVal of r_6 is False, it will not fire. Thus INC terminates after the execution of five rules.

Step	`deltaVals`	`condVals`	`abstract schedule`
1	[[T,T],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_1_1>,<T,T,a_1_2>]
2	[[F,T],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_1_2>]
3	[[F,F],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_2>]
4	[[U,F],U,T,T,T,U]	[T,U,T,T,T,F]	[<T,T,a_3>]
5	[[U,F],U,F,T,T,U]	[T,U,U,T,T,F]	[<T,T,a_4>]
6	[[U,F],U,F,F,T,U]	[T,U,U,T,T,F]	[<T,T,a_5>]
7	[[U,F],U,F,U,F,U]	[T,U,U,T,T,F]	[]

We thus infer that the actual rule execution will also terminate after executing these rules, at most (in fact we can infer that it will execute *precisely* this sequence of rules because no triggered rule has an Unknown event, condition, or delta query). Thus, checking for the k limit during the subsequent actual rule

[6] For reasons of brevity, we use the relational algebra notation for this and subsequent examples, rather than SQL. In this and subsequent examples, all the rules are statement-level rules, unless otherwise stated.

execution can be turned off since we know that this rule execution will terminate.

4 Detecting possible non-termination

Both the actual rule execution and the abstract semantics of INC suffer from the drawback that the **while** loop may fail to terminate. One way of preventing this for the actual rule execution would be to maintain a history of states and to check for the reoccurrence of a past state, where a "state" consists of the current database and current action to be executed.

Due to the size of real databases, this may not be a feasible extension to `execRules`, although work in [7] shows how to improve its efficiency. However, checking for repeating states is also possible in INC and moreover is much less costly in this case. The "state" now consists of the triple `<ev,cv,a>` which is currently at the head of the abstract schedule, and the current truth values of the rule conditions and rule delta queries. The history is a list of such states and is maintained by prefixing the current state to the history on each iteration of the **while** loop. An extra test is added to the **while** loop to test if the current state is equal to a state in the history — note that this is just a simple syntactic equality:

```
history := [];
while s != [] do {
     <ev,cv,a>:= head s;
     s         := tail s;
     if member(<deltaVals,condVals,<ev,cv,a>>,history) then
          return(PossibleNonTermination);
     history  := [<deltaVals,condVals,<ev,cv,a>>] ++ history;
     deltaVals:= INFER_DELTAS(deltaVals,deltaQueries,<ev,cv,a>);
     eventVals:= ... as before ... }
return(DefiniteTermination);
```

There is a bounded number of possible distinct states that `INC` can reach, of the order of $\mathrm{NoOfRules}*3^{\mathrm{NoOfQueries}}$ (since there are NoOfRules rules and NoOfQueries condition and delta queries each of which can take one of three truth values). The extended `INC` function is thus guaranteed to terminate since a repeating state will eventually occur if execution proceeds for long enough. Notice that this is even true for rules that are non-function-free. In contrast, in the actual rule execution, a repeating state may never occur if the rule language is able to create new constants.

Example 2. Consider the following two rules:

Rule	Event	Condition	Action	Delta Query
r_1	$del(R_3)$	$R_8 \bowtie R_4$	$+R_3 \leftarrow R_5$	$R_5 - R_3$
r_2	$ins(R_3)$	$R_5 - R_4$	$-R_3 \leftarrow R_4$	$R_4 \cap R_3$

We trace through an execution of INC in the same way as before:

Step	deltaVals	condVals	abstract schedule
1	[T,T]	[T,T]	[<T,T,a_1>]
2	[F,T]	[T,T]	[<T,T,a_2>]
3	[U,F]	[T,T]	[<T,T,a_1>]
4	[F,U]	[T,T]	[<U,T,a_2>]
5	[U,F]	[T,T]	[<U,T,a_1>]
6	[F,U]	[T,T]	[<U,T,a_2>]

Notice that both rule actions are idempotent. Step 1 shows an initial abstract database state where r_1 has fired. In step 2 r_2 has fired. The two rules continue to fire in turn during steps 3 to 6. In step 3 r_2's action makes r_1's delta query Unknown and in step 4 r_1's action makes r_2's delta query Unknown. The state in step 6 is a repetition of that in step 4 and so INC terminates with a status of `PossibleNonTermination`. It is therefore not safe to ignore the default value of the k limit during the subsequent actual rule execution.

In summary, if INC returns `PossibleNonTermination`, then we perform actual rule execution with the default k limit (k_{def}). Otherwise, if INC returns `DefiniteTermination` after l iterations, then we perform rule execution with the k limit set to l. If $l > k_{def}$, then we will have prevented an unnecessary abort of rule execution. If $l \leq k$ or `PossibleNonTermination` was returned by INC, then running INC will have brought no benefits, but since the cost of INC is low (see Section 5) there will have been negligible impact on system performance. A final point to note is that adding new rules to the rule set is easily accomplished: extra entries just need to be created for these rules in the `condVals` and `deltaVals` arrays and their starting values set to Unknown.

5 Evaluation

Cost and effectiveness of the method. We have extended the PFL active database system [17] with the dynamic rule analysis method described here. The history of abstract database states is stored in main memory as a list of arrays of truth values. The time cost incurred is primarily dependent on the cost of (a) performing the INFER functions and (b) scanning the history for a repeating state. The cost of (a) is negligible. With our present simple encoding of the history, the cost of (b) grows quadratically with the length of the history and on a mid-range Pentium machine reaches 0.5 seconds at a history length of approximately 500 and 1 second at a history length of approximately 1500 Clearly, more sophisticated encodings of the history would improve these times.

The main space cost incurred is by the history. This is bounded by the number of different states that can occur, of the order of $\text{NoOfRules} * 3^{\text{NoOfQueries}}$. This could be very large if NoOfRules is greater than 10 (say). In practice an upper limit, L, will be imposed on history length, determined by the time cost of searching for repeating states[7]. We conjecture setting L to a relatively high

[7] Note that L should not be confused with the default k limit for actual rule execution.

number (2 to 3 orders of magnitude larger than the default k limit) will be sufficient to detect most cases of definite termination, and our experiments with synthetic sets of rules indicate that this is indeed the case. [8]

Relationship to Transaction Execution. We now consider how our dynamic analysis approach integrates with the overall system behaviour. This behaviour can be divided into two modes, transaction mode and rule execution mode. In transaction mode, transactions are submitted to the database and are executed. The truth values of the condition queries and delta queries are maintained in synch with the updates made by the transaction, using the same incremental evaluation techniques as used during INC. If a rule is triggered within a transaction, the system enters rule execution mode. This consists of executing INC followed the actual rule execution. After termination of rule execution, the system reverts to transaction mode again.

Integration with other methods. Extra knowledge about properties of the rule set may be available and it is possible to incorporate this into the INC algorithm. Firstly, we may know from prior static analysis that execution is guaranteed to terminate when certain rules are reached, in which case INC need not continue if it reaches one of these rules. Secondly, we can utilise human-aided analysis in the inference procedure. For instance, rules may be *self disactivating* (i.e. their condition is always made False by their action) and this knowledge can be used for more accurate inferencing. Inferencing can also be made more precise by applying containment and satisfiability tests to determine whether actions will ever have an effect on condition and delta queries. Finally, there is also scope for improving inferencing using knowledge about global properties of rules (e.g. event based stratifications of rule sets as in [6]).

Relationship to SQL3. The rule execution semantics of `execRules` undertake a **semantic, set-oriented** triggering of rules. It is semantic triggering in the sense that rule event queries are delta relations, and rules are triggered if these delta relations are non-empty. In contrast, SQL3's statement-level triggers undertake a **syntactic** triggering of rules i.e. rules are triggered when update statements occur irrespective of whether they cause any change to the database.

Our semantics are easily modified to model execution of SQL3 statement-level AFTER triggers. The condition parts of rules need to be encoded within the bodies of the rule actions and not evaluated within the `for` loop of `execRules`. This causes rule conditions to be evaluated with respect to the database state that the action will be executed on, rather than the database state when the rule was triggered. We use a schedule to store triggered rule actions, rather than SQL3's *trigger execution contexts* (TECs), but the effect is the same. The event parts of rules now need to be event relations (*insEventR* or *delEventR*) rather than delta relations ($\triangle R$ or $\bigtriangledown R$). Incremental inferencing proceeds as with semantic triggering, except now all delta query values are assumed to be always True and event query values will be either True or False (never Unknown).

SQL3's BEFORE triggers are not significant for the purposes of termination

[8] Of course it may be tempting to just use a default k limit set to some high number e.g. 1000, but this would result in wasted execution when the rules are non-terminating.

analysis since they cannot trigger any other trigger. We have also ignored the checking of integrity constraints and have assumed that these will not be violated (for the purposes of termination analysis the worst-case scenario is that no integrity constraints are violated and no aborts occur).

Our rule execution semantics are set-oriented rather than instance-oriented and thus do not directly model the behaviour of SQL3's row-level triggers. However, `execRules` is easily extended to also support row-level triggers, as follows. If a rule `i` has fired and it is a row-level rule then for each tuple `t` $\in$ `eval(eventQueries[i])` a list of actions is placed on the schedule, obtained from `actions[i]` by substituting each occurrence of `eventQueries[i]` in the body of the actions by the singleton set {`t`}.

The question is how should INC be modified, and in particular how many copies of each rule action should be placed on the abstract schedule ? The solution is to dynamically place as many copies as are required until either a repeating state is reached or definite termination can be concluded.

So we recursively invoke INC from the current abstract database state, db_0, with a schedule consisting of 1 copy of the rule action. Either a repeating state will be encountered or definite termination will be inferred. In the former case, we are done and the overall result of INC is possible non-termination. In the latter case we examine if the abstract database state now reached, db_1 say, is the same as db_0. If so, we are done and we can infer definite termination of the execution of this row-level rule and continue processing the rest of the schedule. Otherwise, we recursively invoke INC again from db_1 with a schedule again consisting of 1 copy of the rule action. We continue recursively invoking INC in this way, obtaining a sequence of database states db_0, db_1, db_2, ... db_i, until either possible non-termination results from the i^{th} invocation or db_i is equal to some prior db_j. In the former case the overall result of INC will be possible non-termination. In the latter case we can infer definite termination of the execution of this row-level rule and continue processing the rest of the schedule.

Example 3. Consider the rule set and initial abstract database state of Example 1, but now make rule 5 a row-level rule. Execution is the same as in Example 1 for steps 1-4. At step 5, an action is placed on the schedule, corresponding to a single instance of r_5's action, and INC is recursively invoked with this singleton schedule; at step 6, r_6's condition is false, so the schedule becomes empty and the recursive invocation of INC terminates. INC is then re-invoked recursively, with another instance of r_5's action. Again this recursive invocation terminates, this time with the same state as in Step 6. We thus conclude that execution of the row-level rule r_5 terminates. The overall execution of INC thus terminates. So, checking for the k limit during the subsequent actual rule execution can be turned off since we know that this rule execution will terminate.

Step	`deltaVals`	`condVals`	`abstract schedule`
1	[[T,T],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_1_1>,<T,T,a_1_2>]
2	[[F,T],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_1_2>]
3	[[F,F],T,T,T,T,F]	[T,T,T,T,T,F]	[<T,T,a_2>]

4	[[U,F],U,T,T,T,U]	[T,U,T,T,T,F]	[<T,T,a_3>]
5	[[U,F],U,F,T,T,U]	[T,U,U,T,T,F]	[<T,T,a_4>]
6	[[U,F],U,F,F,T,U]	[T,U,U,T,T,F]	[<T,T,a_5>]
7	[[U,F],U,F,U,F,U]	[T,U,U,T,T,F]	[]
8	[[U,F],U,F,U,T,U]	[T,U,U,T,T,F]	[<T,T,a_5>]
9	[[U,F],U,F,U,F,U]	[T,U,U,T,T,F]	[]

Example 4. Consider the next rule set. Suppose that r_1 is a row-level rule, r_2, r_3 and r_4 are statement-level rules and that r_3 has higher priority than r_2.

Rule	Event	Condition	Action	Deltaval
r_1	$del(R_3)$	$R_8 \bowtie R_4$	$+R_3 \leftarrow R_5 \cap \bigtriangledown R_3$	$(R_5 \cap \bigtriangledown R_3) - R_3$
r_2	$ins(R_3)$	$R_5 - R_3$	$-R_6 \leftarrow R_4$	$R_6 \cap R_4$
r_3	$ins(R_3)$	$R_1 - R_6$	$-R_7 \leftarrow R_4$	$R_7 \cap R_4$
r_4	$del(R_7)$	$true$	$-R_7 \leftarrow R_3 \bowtie R1 \bowtie R_7$	$R_7 \cap (R_3 \bowtie R_1 \bowtie R_7)$

Consider the following execution trace of INC from the stated initial state. Note that the actions of rules r_1, r_2 and r_3 are idempotent. Rule r_1 is assumed to be initially triggered and INC is recursively invoked with a single instance of its action on the schedule. This invocation terminates at step 3. Another recursive invocation of INC is performed at step 4. This time both r_2 and r_3 are triggered and their actions placed on the schedule in order of priority. Execution continues and a repeating state is found at steps 7 and 8. We therefore conclude the set of rules is possibly non-terminating from this initial state:

Step	`DeltaVals`	`condVals`	`abstract schedule`
1	[T,T,T,T]	[T,T,F,T]	[<T,T,a_1>]
2	[F,T,T,T]	[T,U,F,T]	[<T,U,a_2>]
3	[F,F,T,T]	[T,U,U,T]	[]
4	[T,F,T,T]	[T,U,U,T]	[<T,T,a_1>]
5	[F,F,T,T]	[T,U,U,T]	[<T,U,a_3>,<T,U,a_2>]
6	[F,F,F,U]	[T,U,U,T]	[<U,T,a_4>,<T,U,a_2>]
7	[F,F,U,U]	[T,U,U,T]	[<U,T,a_4>,<T,U,a_2>]
8	[F,F,U,U]	[T,U,U,T]	[<U,T,a_4>,<T,U,a_2>]

Remark. If rule r_1 had been statement-level, then execution would have halted at step 3 and definite termination from the initial state concluded.

6 Related Work

Most previous work on termination analysis for active rules has dealt with static analysis. In [1], triggering graphs were first presented and later refined in in [11] and [13], where techniques for removing paths are shown. Generalisations using a method called rule reduction are described in [7]. If one were to extend this method to allow incorporation of knowledge about constraints known to hold in the database (e.g. truth values of conditions), then the result would be quite similar to the approach we have described in this paper. The idea of using abstract interpretation to assist with rule analysis was first presented in [2] and extended to a more complex language in [4]. In [4] complex, and potentially expensive, abstractions are used without any knowledge about the initial database state, whereas here we use a simple, and cheap, abstraction together with some knowledge about the initial database state.

The principal previous work in dynamic analysis is [7], where checking is performed at run-time to see whether a repeating database state has occurred in a history of previous states. In principle, the method is totally precise. However, it is restricted to function-free rules. Also, even with the optimisations presented, it is potentially quite expensive to store the history of database states. Moreover, for real-time applications it is less applicable if one needs to detect loops in a timely manner i.e. before having needlessly executed rules.

Our inferencing depends on deducing new query values from old ones in the presence of updates, and we can make use of previous techniques such as [15, 8, 16, 9, 10]. These techniques were developed for use in query optimisation. Here we have applied them instead to analysis, our focus being on when the values of queries should to be incrementally inferred and with what input, rather than how the inference should be done. Finally, optimisation of triggers based on execution flow is treated in [14], but there it is the execution context of a transaction program, not the rules themselves, which is of interest.

7 Summary and Further Work

We have examined the problem of dynamically enforcing termination of active rules. Rather than aborting rule execution after a fixed number of rule firings, we have described an approach which dynamically infers this limit. The advantage of this is that more terminating rule execution sequences can be allowed to proceed without being aborted prematurely. The algorithms required are cheap since they only operate on synthetic databases, and are thus immediately usable in practical systems. There are a number of directions of further work:

Improving the precision of INC. INC loses precision when condition or delta queries have an Unknown value, which may result in `INC` returning `PossibleNonTermination` where `execRules` would terminate. This can be alleviated in two ways. (a) condition/delta queries that have a current value of Unknown can be evaluated during periods of DBMS inactivity (and thus be updated to True or False), so that the input to the next invocation of INC is made more precise. (b) INC can be interleaved with phases of actual rule execution e.g. if actual rule execution reaches k, rather than giving up and aborting INC can be reinvoked to see more accurate inferences can be made. In particular, if definite termination can now be concluded from the new abstract database state, then the actual rule execution can be allowed to proceed to termination.

Improving the precision of the INFER functions. Rather than just recording the truth value of condition and delta queries, we could record truth values for their sub-queries also. For even more precision, we could use even more detailed information, such as the number of tuples that made a condition true.

Other analysis questions. The thrust of this paper has been towards termination enforcement. However, our techniques are also applicable to other analysis questions that involve reachability e.g. would a given rule eventually be triggered during rule processing on the current database ?

Using definite analysis information for rule optimisation. Suppose we infer from INC that certain rules are guaranteed to have a definitely True or False condition at particular points in the rule execution sequence (e.g. as in Example 1). Then can then use this knowledge to optimise the subsequent actual rule execution sequence by turning off condition evaluation at those points.

References

1. A. Aiken, J. Widom, and J. M. Hellerstein. Static analysis techniques for predicting the behavior of active database rules. *ACM TODS*, 20(1):3–41, 1995.
2. J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard. Abstract interpretation of active rules and its use in termination analysis. In *Proc. 6th ICDT, LNCS 1186*, pages 188–202, 1997.
3. J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proc. of PODS'98.*
4. J. Bailey and A. Poulovassilis. Abstract interpretation for termination analysis in functional active databases. *J. of Intell. Info. Systems*, 12(2/3):243–273, 1999.
5. J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proc. 7th DBPL*, September 1999.
6. E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM TODS*, 21(1), 1996.
7. E. Baralis, Ceri. S., and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE TKDE*, 10(3):353–370, 1998.
8. E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. In *Proc. 2nd RIDS, LNCS 985*, pages 292–308, Athens, 1995.
9. T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *ACM SIGMOD Record*, 27(3):22–27, 1998.
10. T. Griffin, L. Libkin, and H. Trickey. A correction to "Incremental recomputation of active relational expressions". *IEEE TKDE*, 9(3):508–511, 1997.
11. A. Karadimce and S. Urban. Refined triggering graphs: A logic based approach to termination analysis in an active oo database. In *Proc. of ICDE96*, pages 384–391.
12. K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in SQL3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. 1999.
13. S. Y. Lee and T. W. Ling. A path removing technique for detecting trigger termination. In *Proc. 6th EDBT*, pages 341–355, Valencia, 1998.
14. F. Llirbat, F. Fabret, and E. Simon. Eliminating costly redundant computations from SQL trigger executions. In *Proc. ACM SIGMOD*, pages 428–439, 1997.
15. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.
16. D. Quass. Maintenance expressions for views with aggregation. In *Proc. Workshop on Materialised Views: Techniques and Applications*, pages 110–118, 1996.
17. S. Reddi, A. Poulovassilis, and C. Small. PFL: An active functional DBPL. In N. Paton, editor, *Active Rules in Database Systems*, pages 297–308. 1999.